Science of Computer Programming 15 (1990) 119-133
North-Holland

DISTRIBUTED SORTING

H. Peter HOFSTEE, Alain J. MARTIN and Jan L.A. VAN DE SNEPSCHEUT
Computer Science, California Institute of Technology, Pasadena, CA 91125, USA

Revised June 1990

Abstract. In this paper we present a distributed sorting algorithm, which is a variation on exchange sort, i.e., neighboring elements that are out of order are exchanged. We derive the algorithm by transforming a sequential algorithm into a distributed one. The transformation is guided by the distribution of the data over processes. First we discuss the case of two processes, and then the general case of one or more processes. Finally we propose a more efficient solution for the general case.

1. Program notation

For the sequential part of the algorithms, we use a subset of Edsger W. Dijkstra's guarded command language [1]. For (sequential) statements $S0$ and $S1$, statement $S0 \| S1$ denotes their concurrent execution. The constituents $S0$ and $S1$ are then called processes. The statements may share variables (cf. [6]). We transform our algorithms in such a way, however, that the final code contains no shared variables and all synchronization and communication is performed by message passing. The semantics of the communication primitives is as described in [5]. The main difference with Hoare's proposal in [3] is in the naming of channels rather than processes. In [4], the same author proposes to name channels instead of processes in communication commands, but differs from our notation by using one name per channel instead of our two: output command $R!E$ in one process is paired with input command $L?v$ in another process by declaring the pair (R, L) to be a channel between the two processes. Each channel is between two processes only. When declaring (R, L) to be a channel, we write the name on which the output actions are performed first and the name on which the input actions are performed last.

For an arbitrary command A, let cA denote the number of completed A-actions, i.e., the number of times that command A has been executed since initiation of the program's execution. The *synchronization requirement* (cf. [5]) fulfilled by a channel (R, L) is that

$$cR = cL$$

holds at any point in the computation.

Note. It is sometimes attractive to weaken the synchronization requirement by putting some bound on $cR - cL$. This may be a lower bound only, or both a lower and an upper bound. The maximum difference is then called the *slack*, since it indicates how far the synchronized processes can get out of step. The use of a nonzero slack sometimes leads to minor complications in proofs and definitions, and is not pursued here.

The execution of a command results either in the completion of the action or in its suspension when its completion would violate the synchronization requirement. From suspension until completion an action is *pending* and the process executing the action is delayed. We introduce boolean qA equal to the predicate "an A-action is pending". The *progress requirement* states that actions are suspended only if their completion would violate the synchronization requirement, i.e., channel (R, L) satisfies

$$\neg qR \ \vee \ \neg qL$$

The nth R-action is said to match the nth L-action. The completion of a matching pair of actions is called a *communication*. The *communication requirement* states that execution of matching actions $R!E$ and $L?v$ amounts to the assignment $v := E$.

2. A small/large sorter for two bags

Given are two finite, nonempty bags of integers. The integers in the two bags are to be rearranged such that one bag is dominated by the other bag, i.e., no element of the first bag exceeds any element of the second bag. The number of elements of each of the two bags may not be changed.

We use the following notation. The two bags to be sorted are $b0$ and $b1$; their initial values are $B0$ and $B1$ respectively. For bag b, $\#b$ denotes the number of elements in b. Bag union and difference are denoted by $+$ and $-$ respectively. The number of times that a number x occurs in the bag union $b0+b1$ is the number of occurrences of x in $b0$ plus the number of occurrences in $b1$. The number of occurrences of x in $b0-b1$ is the number of occurrences of x in $b0$ minus the number of occurrences in $b1$, and is well-defined only if the latter difference is nonnegative. We do not distinguish between elements and singleton bags.

Postcondition Z of the distributed sorting program is concisely written as follows:

$$Z: \quad \#b0 = \#B0 \ \wedge \ \#b1 = \#B1 \ \wedge$$
$$b0 + b1 = B0 + B1 \ \wedge \ \max(b0) \leq \min(b1)$$

The first two conjuncts express that the size of the two bags is unaffected, the third conjunct expresses that the elements involved remain the same, and the fourth conjunct expresses that $b0$ is dominated by $b1$. Notice that $\max(b0) < \min(b1)$ is a stronger requirement: in fact, it is so strong that it cannot be established in general.

The problem can simply be solved by repeatedly exchanging the maximum element of $b0$ and the minimum element of $b1$ until postcondition Z is established. This amounts to selecting the first three conjuncts of Z as invariant

$$\#b0 = \#B0 \ \wedge \ \#b1 = \#B1 \ \wedge \ b0 + b1 = B0 + B1$$

and the negation of the last conjunct of Z as guard of the repetition. The program is

do $\max(b0) > \min(b1) \rightarrow$

$b0, b1 := b0 + \min(b1) - \max(b0), b1 + \max(b0) - \min(b1)$

od

The invariant is vacuously true upon initialization since then $b0 = B0 \wedge b1 = B1$. The invariant is maintained by the exchange statement, independent of the guard. Upon termination $\max(b0) \leq \min(b1)$ holds, which in conjunction with the invariant implies postcondition Z. We are left with the easy task of proving termination. Let variant function s be the sum of the elements in $b0$ minus the sum of the elements in $b1$. Since $b0$ and $b1$ are finite bags with fixed union, s is bounded from below. On account of the guard, every exchange decreases the sum of the elements in $b0$ and increases the sum of the elements in $b1$, and thereby decreases s. Hence, the loop terminates.

3. Program transformation

We shall now transform the program, under invariance of its semantics, so as to partition it into two sets of (almost) noninterfering statements. We introduce fresh variables both for this purpose and for avoiding repeated evaluation of $\max(b0)$ and $\min(b1)$. When we have two sets of noninterfering statements they can be executed by two processes, which is what we aim at. The interference that remains translates into communication or synchronization actions. Introducing M and m to avoid reevaluation of max and min, and copies LM and rm to reduce interference yields the program in Fig. 1.

Notice that guard $\max(b0) > \min(b1)$ can be rewritten in many ways, including $M > rm$ and $LM > m$. In Fig. 1, we have not made a choice yet, and both rewrites will be used later (which is the reason for not writing down a specific guard here). The bag differences in $b0 + rm - M$ and $b1 + LM - m$ are well-defined since M is an element of $b0$ and m is an element of $b1$. Apart from the concurrent assignment $rm, LM := m, M$ we have partitioned the program into two sets of noninterfering statements. Since the order of noninterfering statements can be swapped freely, we can modify the program slightly so as to group together the actions on $b0$, M, rm and the actions on $b1$, m, LM. We obtain Fig. 2 in which a suggestive layout has been used.

```
M, m := max(b0), min(b1);
rm, LM := m, M;
do guard →
    b0, b1 := b0 + rm − M, b1 + LM − m;
    M, m := max(b0), min(b1);
    rm, LM := m, M
od
```

Fig. 1.

```
        M := max(b0)   ||   m := min(b1);
        rm := m        ||   LM := M;
                do guard →
(b0 := b0 + rm − M; M := max(b0))   ||   (b1 := b1 + LM − m; m := min(b1));
        rm := m        ||   LM := M
                od
```

Fig. 2.

Now, assume that we can split the action $rm := m \parallel LM := M$ into two concurrent parts, X and Y say, such that $cX = cY$ and $\neg qX \vee \neg qY$ hold, and such that the completion of X and Y is equivalent to $rm := m \parallel LM := M$. We may rewrite the program from Fig. 2 into $(p0 \parallel p1)$ as given in Fig. 3. Notice that we have used both ways of rewriting the guard mentioned above.

The correctness of the program in Fig. 3 can be proved in two ways. We may either prove the correctness of the transformation, or we may prove the correctness of the program in Fig. 3 directly. Proving the correctness of the transformation is the more elegant (and slightly easier) of the two. Yet we give a direct proof of the program's correctness, because it comes closer to suggesting the generalization to any number of processes. We postulate that P is an invariant of the distributed program.

$$P: \quad \max(b0) = M = LM \ \wedge \ \min(b1) = m = rm \ \wedge$$

$$\#b0 = \#B0 \ \wedge \ \#b1 = \#B1 \ \wedge \ b0 + b1 = B0 + B1$$

$$
\begin{aligned}
p0 \;\equiv\;& M := \max(b0);\ X;\\
& \textbf{do } M > rm \rightarrow\\
& \quad b0 := b0 + rm - M;\ M := \max(b0);\ X\\
& \textbf{od}\\
p1 \;\equiv\;& m := \min(b1);\ Y;\\
& \textbf{do } LM > m \rightarrow\\
& \quad b1 := b1 + LM - m;\ m := \min(b1);\ Y\\
& \textbf{od}
\end{aligned}
$$

Fig. 3.

What do we mean by claiming that P is an invariant of $(p0 \| p1)$? Both $p0$ and $p1$ contain a loop and by invariant we mean in this case that P holds when both processes have completed the initialization (and no further actions), and that P is maintained if both processes perform one step of the loop. Since initialization and loop body end with action X in $p0$ and action Y in $p1$, and since we have $cX = cY$, this makes sense. Notice that, for example, we do not claim that P holds if $p0$ has completed X, whereas $p1$ has completed Y and also the subsequent update of $b1$. In order to check the invariance, we have to verify:

(a) $\{true\}(M := \max(b0);\ X) \| (m := \min(b1);\ Y)\{P\}$

(b) $P \Rightarrow (M > rm \;\equiv\; LM > m)$

(c) $\{P \wedge M > rm \wedge LM > m\}$
$(b0 := b0 + rm - M;\ M := \max(b0);\ X) \|$
$(b1 := b1 + LM - m;\ m := \min(b1);\ Y)$
$\{P\}$

All three follow from the choice of P and the assumptions on X and Y.

We are left with the task of providing X and Y in terms of the commands that we have at our disposal. Using channels (R, L) and (l, r), with zero slack, we may write:

$$X \;\equiv\; R!M;\ r?rm$$

$$Y \;\equiv\; L?LM;\ l!m$$

Since $cX = cr$ and $cY = cl$ by construction, and since $cr = cl$ by definition, we have $cX = cY$. Actions X and Y may be suspended on either channel, hence

$$qX \;\equiv\; (cR = cr \wedge qR) \vee (cR = cr + 1 \wedge qr)$$

$$qY \;\equiv\; (cL = cl \wedge qL) \vee (cL = cl + 1 \wedge ql)$$

We calculate

$$
\begin{aligned}
& qX \wedge qY \\
= \quad & \{cR = cL,\ cr = cl\} \\
& (cR = cr = cL = cl \wedge qR \wedge qL) \vee \\
& (cR = cr + 1 = cL = cl + 1 \wedge qr \wedge ql) \\
\Rightarrow \quad & \\
& (qR \wedge qL) \vee (qr \wedge ql) \\
= \quad & \{\neg qR \vee \neg qL,\ \neg qr \vee \neg ql\} \\
& false
\end{aligned}
$$

i.e., we have $\neg qX \vee \neg qY$ as required. From the communication requirement it follows that $X \parallel Y$ is equivalent to $rm := m \parallel LM := M$.

The following, more symmetric, version of X and Y also meets the requirements.

$$X \equiv R!M \parallel r?rm$$

$$Y \equiv L?LM \parallel l!m$$

The above two versions are also correct if the slack is positive. The version

$$X \equiv R!M; r?rm$$

$$Y \equiv l!m; L?LM$$

is correct only if the slack is positive.

Observe that the verification of the correctness of the program fits the following pattern. We postulate an invariant and show that it holds in the initial state and is not falsified by an iteration of the loop. We provide a variant function that is bounded from below and decreases with each iteration of the loop. Because the variant function is integer-valued, this implies that the loop terminates. Upon termination we have the truth of the invariant and the falsity of the loop's guard, and we show that this combination implies the postcondition. So much is standard practice in the case of sequential programs. What we add for our distributed programs is the proof of absence of deadlock. Deadlock occurs if one of two processes connected by a channel initiates a communication along the channel and the other process does not. We, therefore, show that mutual communications are initiated under the same condition.

4. More bags

The problem is generalized as follows. Given is a finite sequence of (one or more) finite nonempty bags and a linear array of processes, each of which contains one of the bags and communicates with neighbors in the array to sort the bags in such

a way that each bag is dominated by the next bag in the sequence, and such that the size of the bags remains constant.

The generalized problem is significantly different from the two-bag version in the following sense. Consider sequence ABC of three bags. If A is dominated by B but B is not dominated by C, then an exchange of elements between B and C may cause B to no longer dominate A, i.e., it may necessitate an exchange between A and B. This shows that the process that stores A cannot be terminated when A is dominated by B. The proper thing to do is to terminate a process when the bag it stores is dominated by all bags to the right of it and dominates all bags to the left of it. The two algorithms that follow are both based on exchanging elements of neighboring bags, and termination detection while ensuring progress is the hard part of the problem.

In order to avoid excessive use of subscripts, we use the following notation. For some anonymous process, b is the bag it stores with initial value B, rb is the union of all bags to its right with initial value rB, and lb is the union of all bags to its left with initial value lB. Notice that lB and rB are the empty bag $\emptyset$ for the leftmost and rightmost processes respectively. The required postcondition of the program can be written as a conjunction of terms, one for each process, viz.

$$\max(lb) \leq \min(b) \ \wedge \ \max(b) \leq \min(rb)$$

We find it more attractive to rewrite this into

$$\max(lb) \leq \min(b+rb) \ \wedge \ \max(lb+b) \leq \min(rb)$$

since the first term expresses that domination has been achieved between the union of all bags to the left and the remaining bags, and the second term does so for all bags to the right. The invariant is obtained by introducing a variable for each of the four quantities involved, and by retaining the size restriction on the bags. Hence, the invariant of the distributed program is the conjunction of a number of terms, one for each process. Each such term is

$$\begin{aligned} P: \quad & \max(lb) = LM \ \wedge \ \max(lb+b) = M \ \wedge \\ & \min(rb) = rm \ \wedge \ \min(b+rb) = m \ \wedge \\ & \#b = \#B \ \wedge \ lb+b+rb = lB+B+rB \end{aligned}$$

where $\max(\emptyset) = -\infty$ and $\min(\emptyset) = +\infty$. First we concentrate on the statements that initialize the variables such that P holds. Maxima are simply propagated from left to right, and minima from right to left:

$$(L?LM;\ M := \max(b+LM);\ R!M) \|$$

$$(r?rm;\ m := \min(b+rm);\ l!m)$$

Action $L?LM$ is understood to be $LM := -\infty$ for the leftmost process, and $r?rm$ is understood to be $rm := +\infty$ for the rightmost process. Action $l!m$ is understood to be *skip* for the leftmost process, and $R!M$ is understood to be *skip* for the rightmost

process. These conventions can be implemented with dummy processes next to the two extreme processes, or with conditional statements in the two extreme processes. Next we concentrate on the loop, i.e., we concentrate on maintaining the invariant. Observe that an element from lb should be exchanged with an element from $b+rb$ if $\max(lb) > \min(b+rb)$, i.e., if $LM > m$. Like in the case of two bags, the maximum element from lb is exchanged with the minimum element from $b+rb$. Similarly, the minimum element from rb is exchanged with the maximum element from $lb+b$ if $\max(lb+b) > \min(rb)$, *i.e., if* $M > rm$. This suggests the program shown in Fig. 4.

$(L?LM;\ M := \max(b+LM);\ R!M) \| (r?rm;\ m := \min(b+rm);\ l!m);$

do $LM > m \ \wedge \ M > rm \rightarrow$

$b := b + LM - M + rm - m;$

$(L?LM;\ M := \max(b+LM);\ R!M) \|$

$(r?rm;\ m := \min(b+rm);\ l!m)$

$\square \ LM > m \ \wedge \ M \leq rm \rightarrow$

$b := b + LM - m;$

$(L?LM;\ M := \max(b+LM)) \| (m := \min(b);\ l!m)$

$\square \ LM \leq m \ \wedge \ M > rm \rightarrow$

$b := b + rm - M;$

$(M := \max(b);\ R!M) \| (r?rm;\ m := \min(b+rm))$

od

Fig. 4.

We prove the correctness of this algorithm. Consider two processes that are neighbors in the linear array. Bag $lb+b$ in the left process is bag b in the right process, hence M in the left process has the same value as LM in the right process. Similarly, bag rb in the left process is bag $b+rb$ in the right process, hence rm in the left process has the same value as m in the right process. The left process initiates a communication with the right process if and only if $M > rm$ holds, and the right process initiates a communication with the left process if and only if $LM > m$ holds. Consequently, the two processes initiate their mutual communications under the same condition, which excludes deadlock.

Because of the above-established correspondence between M and rm in one process and LM and m in its righthand neighbor, updating the bags leaves the union of all bags constant *provided* that an element is removed from a bag only if it is contained in that bag. (If this condition is satisfied, then every $+LM$ and $-m$

cancel against a left-neighboring $-M$ and $+rm$.) For example, in order to show that this condition is met for $b := b + LM - M + rm - m$ we prove that M is in $b + LM$, m is in $b + rm$, and $M \neq m$. We are not in the simple situation that we can show that M is in b instead of in $b + LM$: the element M that is being removed from the bag is sometimes the element LM that has just been added. Notice that the order of the bag operations is important: $b := b + LM - M + rm - m$ and $b := b + LM - m + rm - M$ are not equivalent. We prove

(a) $LM > m \wedge M > rm \Rightarrow M \in b + LM \wedge m \in b + rm \wedge M \neq m$

(b) $LM > m \wedge M \leq rm \Rightarrow m \in b + LM$

(c) $LM \leq m \wedge M > rm \Rightarrow M \in b + rm$

Case (a):

$$
\begin{aligned}
& M \in b + LM \wedge m \in b + rm \wedge M \neq m \\
= \quad & \{P\} \\
& \max(lb + b) \in b + \max(lb) \wedge \min(b + rb) \in b + \min(rb) \wedge \\
& \max(lb + b) \neq \min(b + rb) \\
\Leftarrow \quad & \{\max(lb + b) = \max(b + \max(lb)), \min(b + rb) = \min(b + \min(rb))\} \\
& \max(lb + b) > \min(b + rb) \\
\Leftarrow \quad & \{\max(lb + b) \geq \max(lb), P\} \\
& LM > m
\end{aligned}
$$

Case (b):

$$
\begin{aligned}
& m \in b + LM \\
= \quad & \{P\} \\
& \min(b + rb) \in b + \max(lb) \\
\Leftarrow \quad & \{\min(b) \in b\} \\
& \min(b + rb) = \min(b) \\
\Leftarrow \quad & \\
& \max(b) \leq \min(rb) \\
\Leftarrow \quad & \{\max(b) \leq \max(lb + b), P\} \\
& M \leq rm
\end{aligned}
$$

Case (c): similar to case (b).

Termination of the algorithm follows directly from the observation that, in every step of the iteration, the number of inversions is decreased. (An inversion is a pair

of elements from two different bags, where the left element exceeds the right element.) The number of inversions is a natural number and, hence, bounded from below which implies termination. Upon termination we have a state that satisfies both the invariant and the negation of all three guards. We, therefore, have

$$P \wedge LM \leq m \wedge M \leq rm$$

$$\Rightarrow$$

$$\max(lb) \leq \min(b + rb) \wedge \max(lb + b) \leq \min(rb)$$

upon termination, which is the required postcondition.

Notice that the algorithm is not correct if the last two guards are weakened to $LM > m$ and $M > rm$ respectively. It is then possible for elements to be removed from a bag of which they are not an element, implying that the union of all bags is not constant.

Statement $M := \max(b + LM)$ does not change M in the second guarded command, and may, therefore, be omitted. Similarly for $m := \min(b + rm)$ in the third guarded command.

5. A more efficient solution

The invariant proposed in the previous section was easy to guess (and understand), and led to a simple program. On closer inspection, however, it turns out that the program is not very efficient. Each step of the loop contains a construct for propagating maxima from left to right, and minima from right to left. This propagation requires time proportional to the number of bags, making the execution time of the whole program quadratic in the number of bags. Operationally speaking, the processes are suspended most of the time on communications of global extremes. It seems to be more attractive to perform some exchanges of local extremes between neighbors in the mean time: we may hope to obtain a program whose execution time is linear in the number of bags instead of quadratic. This idea is not easily translated into a program, mainly because detecting the end of "the mean time" is nontrivial. A similar effect, however, can be obtained in a different way. Exchanges of local extremes between neighbors may be performed while, in passing, global extremes are computed. The global extremes can be computed by some sort of approximation technique. Formally, this amounts to weakening the invariant from $LM = \max(lb)$, to $LM \geq \max(lb)$, and $rm = \min(rb)$ to $rm \leq \min(rb)$. If we stick to the terms $M = \max(b + LM)$ and $m = \min(b + rm)$, as well as the other terms, then the conjunction of $LM \leq m$ and $M \leq rm$ and the invariant implies the postcondition. Hence, the weaker invariant is still sufficiently strong.

If we aim at a program whose structure is similar to the program in the previous section, we have a loop in which each step corresponds to a communication with the left neighbor, or with the right neighbor, or both. Deadlock is avoided if neighbors

initiate their mutual communications under the same condition. Since only approximations of global extremes are locally available, we cannot simply use $LM > m$ and $M > rm$. Since LM is obtained from the, previously communicated, left neighbor's M, the left neighbor is to initiate a communication based on the previous value of M, say PM. This leads to invariant Q and to the program of Fig. 5.

$$Q: \quad LM \geq \max(lb) \ \wedge \ PM \geq M = \max(b + LM) \ \wedge$$
$$rm \leq \min(rb) \ \wedge \ pm \leq m = \min(b + rm) \ \wedge$$
$$\#b = \#B \ \wedge \ lb + b + rb = lB + B + rB$$

Notice that, due to the exchange of local extremes between neighbors rather than the propagation of global extremes, it may be necessary to replace two elements from the local bag. Hence, this algorithm is applicable only to the case in which each bag (except for the leftmost and rightmost bags) contains at least two elements.

We prove the correctness of this algorithm. Consider two processes that are neighbors in the linear array. We show that PM and rm in the left process have the same value as LM and pm in the right process. Initially we have $PM = +\infty$ and $rm = -\infty$ in the left process (since its rb is nonempty). Similarly we have $LM = +\infty$ and $pm = -\infty$ in the right process. The four variables are assigned a new value only when the two processes communicate with each other. The relevant statements are

$$r?(y, rm) \| R!(\max(b), M); \ PM := M$$

in the left process, and

$$L?(x, LM) \| l!(\min(b), m); \ pm := m$$

in the right process. Inspection reveals that both PM and LM are assigned the value M, and that both rm and pm are assigned the value m. Hence, the correspondence between the variables is maintained. Consequently, the two processes initiate their mutual communications under the same condition, which excludes deadlock.

Next we show that the operations on b do not falsify the invariant. Inspection of the communication statements (as in the paragraph above) reveals that $\max(b)$ and y in the left process correspond to x and $\min(b)$ in the right process. Hence the updates of the bags are performed under the same condition and change neither the union of the bags nor the size of each bag. Notice that the assumption $\#b \geq 2$ is essential here.

In the same vein the invariance of $LM \geq \max(lb)$ and $PM \geq M = \max(b + LM)$ may be proved.

It remains to prove termination. To that end we strengthen the invariant to express that M is a very good approximation of $\max(lb + b)$. In fact, we have either $M = +\infty$ or $M = \max(lb + b)$. We can even prove that also $M = \max(b)$ holds in the latter case. This expresses the (strong) property that the largest value of $lb + b$ resides in bag b, and that the second largest value of $lb + b$ resides either in b or in the left-neighbor's bag, etc. Furthermore, we show that $M = +\infty$ does not persist too long. More precisely we show that, in the process which has k other processes to

```
if lb = ∅ → LM := −∞ □ lb ≠ ∅ → LM := +∞ fi;
if rb = ∅ → rm := +∞ □ rb ≠ ∅ → rm := −∞ fi;
M, PM, m, pm := max(b + LM), +∞, min(b + rm), −∞;
do LM > pm ∧ PM > rm →
    L?(x, LM)‖l!(min(b), m)‖r?(y, rm)‖R!(max(b), M);
    if x > min(b) ∧ max(b) > y → b := b − min(b) − max(b) + x + y
    □ x > min(b) ∧ max(b) ≤ y → b := b − min(b) + x
    □ x ≤ min(b) ∧ max(b) > y → b := b − max(b) + y
    □ x ≤ min(b) ∧ max(b) ≤ y → skip
    fi;
    M, PM, m, pm := max(b + LM), M, min(b + rm), m
□ LM > pm ∧ PM ≤ rm →
    L?(x, LM)‖l!(min(b), m);
    if x > min(b) → b := b − min(b) + x
    □ x ≤ min(b) → skip
    fi;
    M, m, pm := max(b + LM), min(b + rm), m
□ LM ≤ pm ∧ PM > rm →
    r?(y, rm)‖R!(max(b), M);
    if max(b) > y → b := b − max(b) + y
    □ max(b) ≤ y → skip
    fi;
    M, PM, m := max(b + LM), M, min(b + rm)
od
```

Fig. 5.

its left, $M = \max(b) = \max(lb + b)$ holds after k iterations of the loop. We postulate that

$$LM = PM = M = +\infty \ \vee$$

$$(+\infty > LM \geq \max(lb) \ \wedge \ PM \geq M = \max(b) = \max(lb + b))$$

is an invariant, and we verify this claim. Initially $M = +\infty$ holds in every process except in the leftmost process ($k = 0$) in which $M = \max(b) = \max(lb + b)$. If $M = +\infty$ then the process initiates a communication to the left (since $M = +\infty$ implies $LM = +\infty$, and $pm < +\infty$). The relevant statements are

$$L?(x, LM)\ldots;$$

$$\textbf{if } x > \min(b)\ldots \rightarrow b := b - \min(b) + x \ldots \textbf{fi};$$

$$M := \max(b + LM)$$

together with

$$R!(\max(b), M)$$

in the left process. If $M = +\infty$ holds in the left process prior to this step, then $LM = M = +\infty$ holds in the right process after this step. If $M = \max(b) = \max(lb + b)$ holds in the left process prior to this step, then the statement $L?(x, LM)$ in the right process leads to $x = LM = \max(lb)$. Hence, the updates of b and M lead to $M = \max(b) = \max(lb + b)$ in the right process, one iteration after this relation has been established in its left neighbor. Notice that the update of the bag in the left neighbor process may falsify $LM = \max(lb)$, but $LM \geqslant \max(lb)$ is maintained. As a result, in each process we have $M = \max(b) = \max(lb + b)$ after a number of steps equal to the number of processes. Similarly, $m = \min(b) = \min(b + rb)$ holds. When this state has been reached it is not guaranteed that the variant function from the previous two sections is decreased with every iteration of the loop. That variant function contained the bags only, and it is possible that no bag is changed by an iteration of the loop. However, if in this state the bag is not changed then it is the last iteration of the loop: if, for example, $LM > pm$ and $x \leqslant \min(b)$ then $LM = x \leqslant \min(b) = m$, and pm is set to m, thereby falsifying $LM > pm$ which excludes further iterations containing a communication to the left.

Upon termination we have the invariant and the negation of the guards

$$Q \ \wedge \ LM \leqslant pm \ \wedge \ PM \leqslant rm$$

$$\Rightarrow$$

$$\max(lb) \leqslant LM \leqslant pm \leqslant rm = \min(b + rm) \ \wedge$$

$$\max(b + LM) = M \leqslant PM \leqslant rm \leqslant \min(b)$$

$$\Rightarrow$$

$$\max(lb) \leqslant \min(b + rb) \ \wedge \ \max(lb + b) \leqslant \min(rb)$$

which is the required postcondition.

The time complexity of the present solution is linear in the number of bags, N say. Thus, we have gained a factor of N at the expense of sending two integers per communication instead of one, and the addition of two integer variables per process.

If each bag contains k elements, the number of iterations is $N \cdot k$ in the worst case. Assuming that the operations on a bag are $O(\log(k))$ each, this implies that the worst case time complexity is $O(N \cdot k \cdot \log(k))$.

In this program the guards of the second and third alternative of the loop may be weakened to $LM > pm$ and $PM < rm$ respectively, without falsifying the invariant. It has the advantage that the program may be simplified (by omitting the first alternative) and that the requirement $\#b \geq 2$ may be weakened to $\#b \geq 1$, but it has the distinct disadvantage that the program does not necessarily terminate: if both guards are true it is possible that selection of one of the alternatives does not change the state in either of the two processes involved. If fair selection of the alternatives is postulated, then one can show that the variant function decreases eventually, which implies that the program terminates eventually.

6. Conclusion

We have presented this paper as an exercise in deriving parallel programs. First, a sequential solution to the problem is presented which is subsequently transformed into a parallel solution. Next, extra variables and communication channels are introduced. Finally, the invariant is weakened. The transformation steps are not automatic in the sense that absence of deadlock had to be proved separately.

The resulting algorithms have some of the flavor of odd-even transposition sort. They are, however, essentially different in two respects. In every step of the loop in odd-even transposition sort, a process communicates with only one of its two neighbors, whereas in every step of the loop of our algorithms a process communicates with both its neighbors (as long as necessary). The other difference is that our algorithms are "smooth" (cf. [2]) in the sense that the execution time is much less for almost-sorted arrays than for hardly-sorted arrays, with a smooth transition from one to the other behavior. This is due to the conditions under which processes engage in communications.

Acknowledgement

We are grateful to Johan J. Lukkien for many discussions during the design of the algorithms, and to Wayne Luk for helpful remarks on the presentation.

References

[1] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[2] E.W. Dijkstra, Smoothsort, an alternative for sorting in situ, *Sci. Comput. Programming* 1 (1982) 223-233.

[3] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.
[4] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science (Prentice-Hall, Englewood Cliffs, NJ, 1985).
[5] A.J. Martin, An axiomatic definition of synchronization primitives, *Acta Inform.* **16** (1981) 219-235.
[6] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Inform.* **6** (1976) 319-340.